Advanced Java Week 3 - Day04

Agenda:

1) JAVA I/O API - Streams
2) Character Streams and Binary Streams
3) File Handling(Binay Stream and Character Stream based)
4) SERIALIZATION - DESERIALIZATION
5) Collection Pipeline Processing
6) Streams - Java 8 Feature
7) Stream - Intermediate & Terminal Operations
8) Reduction
9) Collector
10) Date Time API - Java8 Feature

JAVA I/O API - Streams

Java provides a rich set of API under the package java.io , using which one can perform File Handling.

File Handling or the I/O operation usually used to represent Input/Output Operation over the files, can be implemented with the help of java.io package.

I/O operations are typically carried out on the Streams, which in turn facilitates the flow of this data into files.

“Streams” are flow of Data which are basically of 2 types **Input Stream** & **Output Stream**, Input Stream is the Stream which facilitates the flow of Data into the Application , the flow of data that is read from the file , whereas the Output Stream is the Stream which facilitates the flow of Data from the application into the file.
Hence, when we need to read from a file, we use InputStream and when we need to write we use OutputStream.

Character Streams and Binary Streams

Java provides rich API to deal with these Streams, Streams are again categorized into 2 categories

1) Character Stream
2) Byte Stream

Character Streams allow the data flow in the character format and Byte Stream allows the data flow in the Binary format.
Depending upon the type of Data that we want to read/write , we can use the corresponding type of Stream, for example when we need to write/read textual data we can use the Character Streams & when we want to write/read some binary data for example audio, video or some image based data, then we can use Binary Stream.

These streams are also called Node Streams, these Node streams can be of the Type files, memory or thread based Node Streams.
In this chapter we are dealing with only File Streams.

File Handling:

I/O Fundamentals

- A stream can be thought of as a flow of data from a source or to a sink.
- A source stream initiates the flow of data, also called an input stream.
- A sink stream terminates the flow of data, also called an output stream.
- Sources and sinks are both node streams.
- Types of node streams are files, memory, and pipes between threads or processes.

I/O Fundamentals (Contd.)

- The fundamental stream classes are:

Stream	Byte Streams	Character Streams
Source streams	InputStream	Reader
Sink streams	OutputStream	Writer

Processing Streams

- Processing Streams are Node Streams that use filters in between while transferring the data.
- Various types of Processing Streams are:

Type	Character Streams	Byte Streams
Buffering	BufferedReader BufferedWriter	BufferedInputStream BufferedOutputStream
Filtering	FilterReader FilterWriter	FilterInputStream FilterOutputStream
Converting between bytes and character	InputStreamReader OutputStreamWriter	
Performing object serialization		ObjectInputStream ObjectOutputStream
Performing data Conversion		DataInputStream DataOutputStream
Counting	LineNumberReader	LineNumberInputStream

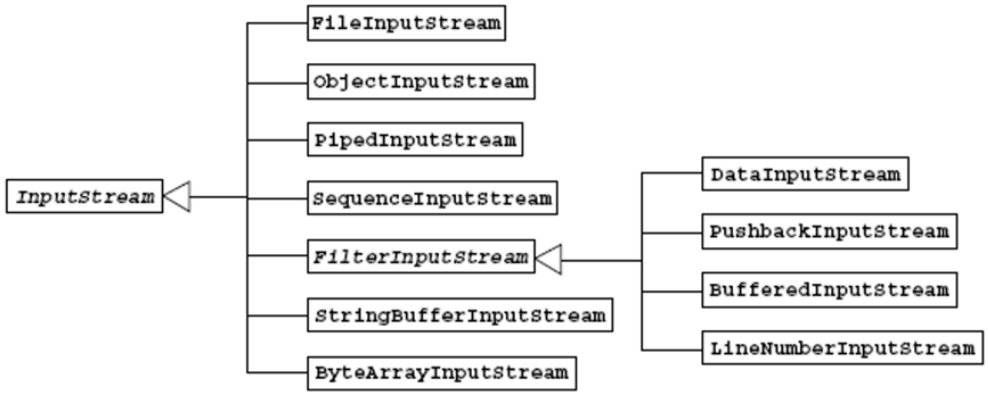
Node Streams

- Various types of Character and Byte Stream classes are:

Type	Character Streams	Byte Streams
File	FileReader FileWriter	FileInputStream FileOutputStream
Memory: array	CharArrayReader CharArrayWriter	ByteArrayInputStream ByteArrayOutputStream
Memory: string	StringReader StringWriter	N/A
Pipe	PipedReader PipedWriter	PipedInputStream PipedOutputStream

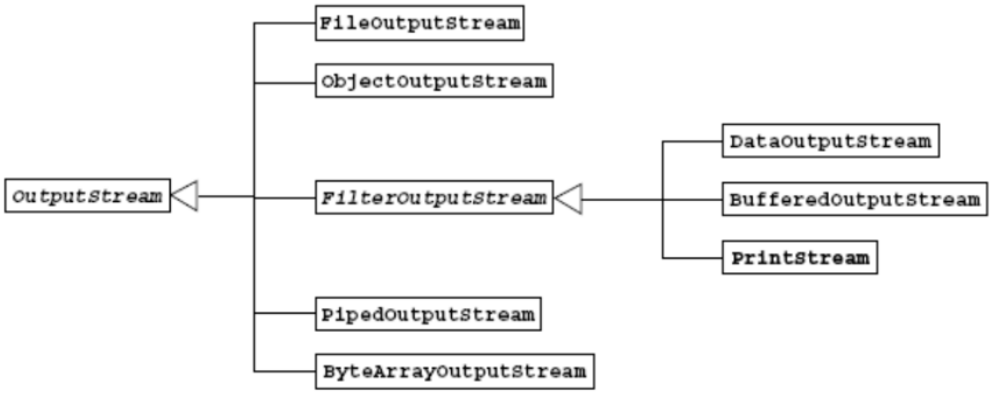
InputStream

- The InputStream Class Hierarchy:



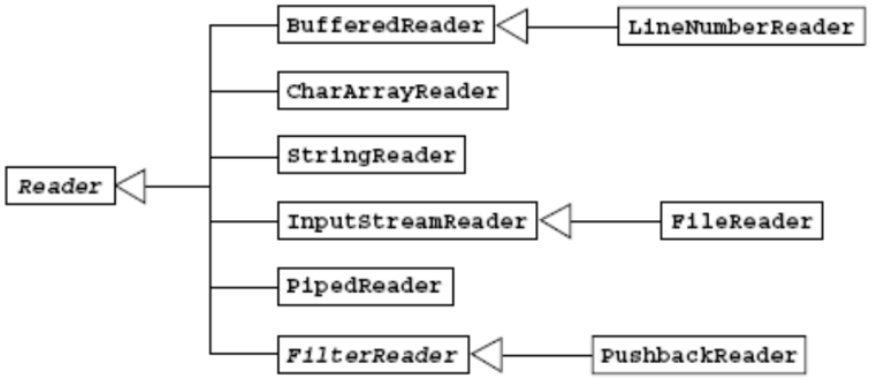
OutputStream

- The OutputStream Class Hierarchy:



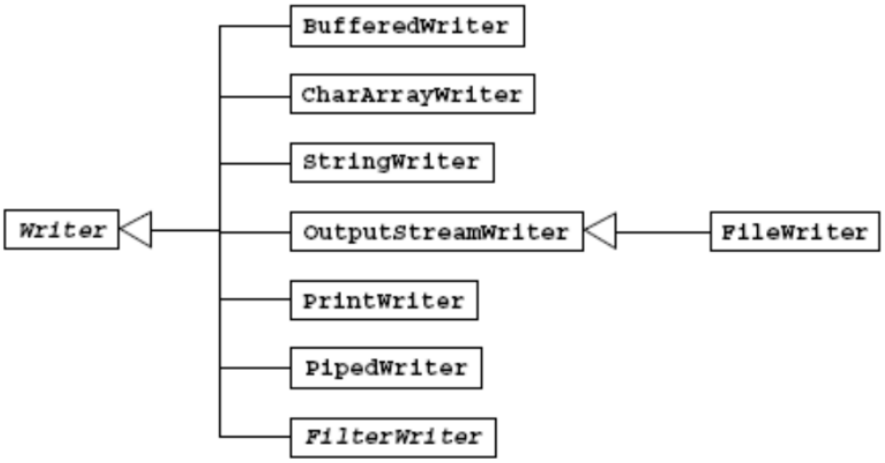
Reader Class

- The Reader Class Hierarchy:



Writer Class

- The Writer Class Hierarchy:



InputStream & OutputStream are 2 abstract classes provided by java.io package , used to process the Binary Streams.
Reader & Writer are 2 abstract classes provided by java.io package , used to process Character Streams. Here we need to restrict the process to only Read or Write

Though we are not discussing all those inherited classes of the above said abstract classes we can at least touch upon a few.

Binary Streams

FileOutputStream & FileInputStream :
Basic writing and reading into Binary streams can be made possible using these 2 classes.

Example Code: Writing

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class BinaryStreamReadWriter {

	byte[] myBytes = new byte[100];
	String str = "Welcome to Streams...";
	//User Defined Method...
	public void writeToBinaryStream()
	{
		try
		{
			myBytes = str.getBytes();//getBytes() function would convert string into bytes
			FileOutputStream fos = new FileOutputStream("Customer.txt");
			fos.write(myBytes);
			fos.flush();
			fos.close();
			System.out.println("We Have written into a file Successfully...");
		}
		catch(FileNotFoundException fne)
		{
			fne.printStackTrace();
		}
		catch(IOException ioe)
		{
			ioe.printStackTrace();
		}

	}


	public static void main(String[] args) {
		// TODO Auto-generated method stub
		BinaryStreamReadWriter bsrw = new BinaryStreamReadWriter();
```

```
                bsrw.writeToBinaryStream();

        }

}
```

Example Code: Reading

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class BinaryStreamReader {

        FileInputStream fis;
        byte[] readBytes = new byte[100];
        public void readFromBinaryStream()
        {

                try
                {

                        fis = new FileInputStream("Customer.txt");
                        fis.read(readBytes);
                        fis.close();
                        String str = new String(readBytes);
                        System.out.println("The Data Read is "+str);

                } catch (FileNotFoundException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                } catch(IOException ioe)
                {
                        ioe.printStackTrace();
                }

        }

        public static void main(String[] args) {
                // TODO Auto-generated method stub
                BinaryStreamReader bsr = new BinaryStreamReader();
                bsr.readFromBinaryStream();
        }

}
```

DataOutputStream & DataInputStream :

When we wish to maintain reading/writing activity with the respective data type
These 2 classes are used to write and read the data in primitive data type formats in a portable manner.

Example Code:

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class DataStreamClass {

	DataOutputStream dos;
	DataInputStream dis;

	public void writeReadData()
	{
		try {

			dos = new DataOutputStream(new FileOutputStream("datafile"));
			dos.writeDouble(2345.56);
			dos.writeUTF("welcome to datastream");
			dos.writeBoolean(true);
			dos.writeInt(10000);
			dos.flush();
			dos.close();

		} catch (FileNotFoundException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		} catch(IOException ioe)
		{
			ioe.printStackTrace();
		}
	}

	public void readData()
	{
		try {
```

```
                    dis = new DataInputStream(new FileInputStream("datafile"));
                    System.out.println("The Double Data is "+dis.readDouble());
                    System.out.println("The UTF Data is "+dis.readUTF());
                    System.out.println("The Boolean Data is "+dis.readBoolean());
                    System.out.println("The Integer Data is "+dis.readInt());
            } catch (FileNotFoundException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
            } catch(IOException ioe)
            {
                    ioe.printStackTrace();
            }
    }
    public static void main(String[] args) {
            // TODO Auto-generated method stub
            DataStreamClass dsc = new DataStreamClass();
                    dsc.writeReadData();
                    dsc.readData();
    }

}
```

In the above program we write the data in the double,String,boolean and int data into a file using DataOutputStream and read the same in the same sequence from the file using DataInputStream classes and print the same on the console..

BufferedOutputStream & BufferedInputStream:

These 2 classes enables us to write and read using buffers and also we can specify the buffer size, Buffered reading and writing is considered to be efficient IO operations

Example :BufferedWriting

```
import java.io.BufferedOutputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class BufferedStreamByteWriter {

    BufferedOutputStream bos;
    String str = "we are writing bytes thru buffer to our file";
    byte myBytes[] = new byte[100];
    public void writeToByteStream()
    {
            try
```

```
                {
                        myBytes = str.getBytes();
                        bos = new BufferedOutputStream(new FileOutputStream("Supplier.txt"));
                //      bos = new BufferedOutputStream(new FileOutputStream("xyz.txt"),1000);
                        bos.write(myBytes);
                        bos.flush();
                        bos.close();
                        System.out.println("We Have written successfully into file using Buffer thru
Streams...");
                } catch (FileNotFoundException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                } catch(IOException ioe)
                {
                        ioe.printStackTrace();
                }
        }
        public static void main(String[] args) {
                // TODO Auto-generated method stub
                BufferedStreamByteWriter bsbw = new BufferedStreamByteWriter();
                bsbw.writeToByteStream();

        }

}
```

Example :BufferedReading

```
import java.io.BufferedInputStream;
import java.io.IOException;

public class BufferedStreamReaderSample {

        BufferedInputStream bis;
        byte myBytes[] = new byte[100];
        String str;
        public void readThruBufferFromKeyBoard()
        {
                bis = new BufferedInputStream(System.in);
                System.out.println("Enter a String ...");
                try
                {
                        bis.read(myBytes);
                        str = new String(myBytes);
                        System.out.println("You Entered :"+str);
                        bis.close();
```

```
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

	}
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		BufferedStreamReaderSample bsrs = new BufferedStreamReaderSample();
		bsrs.readThruBufferFromKeyBoard();

	}

}
```

Example : BufferedWriting to Console

```
import java.io.BufferedOutputStream;
import java.io.IOException;

public class BuferedStreamWriterSample {

	BufferedOutputStream bos;
	String str = "We are writing into Stream using Buffer...";
	byte myBytes[] = new byte[100];
	public void writeThruBufferToMonitor()
	{
		myBytes = str.getBytes();
		bos = new BufferedOutputStream(System.out);
		try
		{
			bos.write(myBytes);//Expect this content to be shown on Monitor
			bos.flush();
			System.out.println("We Have Written onto StdO/P File thru Buffer..");
			bos.close();
		//	System.out.println("We Have Written onto StdO/P File thru Buffer..");
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

	}
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		BuferedStreamWriterSample bsws = new BuferedStreamWriterSample();
		bsws.writeThruBufferToMonitor();
```

```
	}

}
```

Example : BufferedReading from Keyboard

```
import java.io.BufferedInputStream;
import java.io.IOException;

public class BufferedStreamReaderSample {

	BufferedInputStream bis;
	byte myBytes[] = new byte[100];
	String str;
	public void readThruBufferFromKeyBoard()
	{
		bis = new BufferedInputStream(System.in);
		System.out.println("Enter a String ...");
		try
		{
			bis.read(myBytes);
			str = new String(myBytes);
			System.out.println("You Entered :"+str);
			bis.close();
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

	}
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		BufferedStreamReaderSample bsrs = new BufferedStreamReaderSample();
		bsrs.readThruBufferFromKeyBoard();

	}

}
```

Character Streams

READER - WRITER

Example: Writer

```
import java.io.FileWriter;
import java.io.IOException;

public class MyFileWriter {

	FileWriter fw;
	String str = "We are writing text into Char Stream";
	public void writeToCharStream()
	{
		try
		{
			fw = new FileWriter("employees.txt");
			fw.write(str);
			fw.flush();
			fw.close();
			System.out.println("We Have written Successfully into char stream");
		} catch (IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

	}
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		MyFileWriter mfw = new MyFileWriter();
		mfw.writeToCharStream();
	}

}
```

Example : Reader

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class MyFileReader {

	FileReader fr;
	int myChar;
	public void readFromFile()
```

```
	{
		try
		{
			fr = new FileReader("employees.txt");
			while((myChar = fr.read()) != -1)
			{
				System.out.print((char)myChar);
			}
			fr.close();
		} catch (FileNotFoundException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		} catch(IOException ioe)
		{
			ioe.printStackTrace();
		}
	}
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		MyFileReader mfr = new MyFileReader();
		mfr.readFromFile();

	}

}
```

Example: BufferedReader / BufferedWriter

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class BufferedCharWriterReader {

	BufferedWriter bw;
	BufferedReader br;
	String str = "We are writing chars into Dealers File thru Buffer";
	public void writeCharsThruBuffer()
	{
		try {
			bw = new BufferedWriter(new FileWriter("Dealer.txt"));
```

```
                bw.write(str);
                bw.flush();
                bw.close();
                System.out.println("We Have written successfully chars into file thru buffer...");

            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

        }
        public void readCharsThruBuffer()
        {
            String str;
            try {
                br = new BufferedReader(new FileReader("Dealer.txt"));
                str = br.readLine();
                br.close();
                System.out.println("The Data Read thru buffer is "+str);
            } catch (FileNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch(IOException ioe)
            {
                ioe.printStackTrace();
            }

        }
        public static void main(String[] args) {
            // TODO Auto-generated method stub
            BufferedCharWriterReader bcr = new BufferedCharWriterReader();
            //bcr.writeCharsThruBuffer();
            bcr.readCharsThruBuffer();
        }

}
```

SERIALIZATION - DESERIALIZATION

Serialization is a process of converting the state of an Object into Stream and Deserializing is a reverse process of converting back the respective Stream into an Object.
Serialization helps in securing the data in such a way that the Objects so serialized would not be tampered during the transit through Streams.

Advantages of Serialization:

Serialization is advantageous during the following tasks

1) Saving/Persisting Objects
2) Transferring Objects over the network

The Classes which are to be Serialized have to implement **Serializable** interface, also in a class those data members whose state need not be Serialized can be declared **transient** which indicates that, these data members need not be Serialized

Serial Version UID: The JVM generates a version number with each serializable class. This id is used to verify the saved and loaded objects have the same attributes and thus are compatible with serialization

.

java.io provides the following classes to Serialize and Deserialize objects.
ObjectOutputStream
ObjectInputStream

Example : Serialization & DeSerialization
Class to be Serialized

```
import java.io.Serializable;
public class Employee implements Serializable{

	String employeeId;
	String employeeName;
	String employeeAddress;
	String employeePhone;
	int employeeSalary;
	/*transient*/ float empTax;
	public Employee() {
		super();
	}
	public Employee(String employeeId, String employeeName, String employeeAddress,
String employeePhone,
			int employeeSalary, float empTax) {
```

```
        super();
        this.employeeId = employeeId;
        this.employeeName = employeeName;
        this.employeeAddress = employeeAddress;
        this.employeePhone = employeePhone;
        this.employeeSalary = employeeSalary;
        this.empTax = empTax;
}
public String getEmployeeId() {
        return employeeId;
}
public void setEmployeeId(String employeeId) {
        this.employeeId = employeeId;
}


public String getEmployeeName() {
        return employeeName;
}
public void setEmployeeName(String employeeName) {
        this.employeeName = employeeName;
}
public String getEmployeeAddress() {
        return employeeAddress;
}
public void setEmployeeAddress(String employeeAddress) {
        this.employeeAddress = employeeAddress;
}
public String getEmployeePhone() {
        return employeePhone;
}
public void setEmployeePhone(String employeePhone) {
        this.employeePhone = employeePhone;
}
public int getEmployeeSalary() {
        return employeeSalary;
}


public void setEmployeeSalary(int employeeSalary) {
```

```
		this.employeeSalary = employeeSalary;
	}
	public float getEmpTax() {
		return empTax;
	}
	public void setEmpTax(float empTax) {
		this.empTax = empTax;
	}

	@Override
	public String toString() {
		return "Employee [employeeId=" + employeeId + ", employeeName=" +
employeeName + ", employeeAddress="
				+ employeeAddress + ", employeePhone=" + employeePhone + ",
employeeSalary=" + employeeSalary
				+ ", empTax=" + empTax + "]";
	}
}
```

Serialization:

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class SerializerClass {

	ObjectOutputStream oos;
	public void serializeToFile()
	{
		try
		{
			oos = new ObjectOutputStream(new
FileOutputStream("myEmployee.txt"));
```

```
                        Employee e1 = new
Employee("E001","Harsha","RTNagar","9838838833",10000,12.34f);
                        oos.writeObject(e1);
                        oos.flush();
                        oos.close();
                        System.out.println("We Have Serialized Employee Object
successfully...");

                } catch (FileNotFoundException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
        public static void main(String[] args) {
                // TODO Auto-generated method stub
                SerializerClass sc = new SerializerClass();
                sc.serializeToFile();
        }

}
```

DeSerialization:

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.ObjectInputStream;

public class DeSerializerClass {

        ObjectInputStream ois;

        Employee e;
        public void deSerializeFromFile()
        {
                try {
                        ois = new ObjectInputStream(new FileInputStream("myEmployee.txt"));
                        e = (Employee)ois.readObject();
```

```
                System.out.println("The OBject Read "+e);
                ois.close();
            } catch (FileNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            } catch(ClassNotFoundException cnfe)
            {
                cnfe.printStackTrace();
            }

    }
    public static void main(String[] args) {
            // TODO Auto-generated method stub

            DeSerializerClass dsc = new DeSerializerClass();
            dsc.deSerializeFromFile();

    }

}
```

Collection Pipeline Processing

Since we have just completed understanding IO Streams and are venturing into Collection Streams, learners should not get confused between these 2 Streams. These 2 are in different contexts.
IOStreams deal with flow of data in character or binary format used in File Handling, whereas Collection Streams deals with Collections of objects in the form of a structure - Stream , on which we perform multiple operations to get analyzed and summarized results.

Collection Pipeline : Involves sequencing a stream of objects , over which the filter & Pipes operations are applied upon.
Collection Pipeline introduces **Function Composition** and **Collection PipeLine**, two functional style patterns that we can combine to iterate collections in our code. In functional programming, it is common to sequence complex operations through a series of smaller modular functions or operations. The series is called a **composition of functions**, or a **function composition**. When a collection of data flows through a function composition, it becomes a collection pipeline. **Function Composition** and **Collection Pipeline** are two design patterns frequently used in functional-style programming.

In this process there are two types of operations that are carried out, initially **INTERMEDIATE** operations are carried out on the Stream of objects , which may involve **filtering , mapping , limiting & distinguishing** , based on these , finally **TERMINAL** operations are to be performed on the resultant Stream data, which may primarily focus on summarizing operations like **reducing, collecting & counting,**

Applications:
Collection Pipeline pattern can be used when

1) One wants to perform a sequence of operations where one operation's collected output is fed into another
2) When a lot of statements/operations are used in the code.
3) When a lot of looping is used in the codes for some operations.

Streams - Java 8 Feature

- **Stream** is the **structure** for processing a **collection** in **functional** style.
- Original collection is not modified.
- Streams may be processed only once. After getting an output you can not use it again.
- Supports sequential and parallel aggregate operations.

Streams – Intermediate & Terminal Operations

Intermediate operations yield a new Stream and you may pipeline(chain) the Operations
Terminal operations yield a result other than a stream: A list, Integer etc.After a terminal operation, you can not use the stream again and you can not pipeline another function.

Streams – Intermediate Operations

Types

Stateful Intermediate Operations : Stateful intermediate operations are those which maintain information from a previous invocation internally(aka state) to be used again in a future invocation of the method.

Stateless Intermediate Operations: Stateless intermediate operations are the opposite of stateful and do not store any state across passes.

Stream Operations	GOAL	Input
filter	**Filter items according to a given predicate**	**Predicate**
map	**Processes items and transforms**	**Function**
limit	**Limits the results**	**int**
sorted	**Sort items inside stream**	**Comparator**
distinct	**Remove duplicate items according to equals method of the given type**	

Intermediate Operation Sample1:

```
public void functional()
{
  //Sum of squares of integers 1 to 5
          List <Integer> numbers = Arrays.asList(1,2,3,4,5);
          Stream <Integer> numberStream = numbers.stream();
          Stream <Integer> oddNumberStream = numberStream.filter(x -> x % 2 ==1);
          Stream <Integer> squaredNumberStream = oddNumberStream.map(x -> x * x);
           int sum = squaredNumberStream.reduce(0, (x,y) -> x + y);
           System.out.println("The sum of squares is "+sum);
}
 public static void main(String[] args)
 {
  // TODO Auto-generated method stub
  functional();
```

```
// Use Objects to invoke functions-class part not shown
}
```

The above program would print the sum of the squares of the odd numbers between 1 to 5

Here numbers.stream() would transfer all the numbers from 1 to 5 in numberStream

Filter function would filter only oddNUmbers into oddNumberStream

Map function would only map the square of those oddNumbers into squareNumberStream and reduce would pick from 0 all numbers and sum it up.

Intermediate Operation Sample2:

Class Programmer

```
package com.core;
//For more info on Java Data & Time java 8 API refer Notes on the Same, in the same document
import java.time.LocalDate;
import java.time.Month;
import java.util.Arrays;
import java.util.List;
public class Programmer {

public static enum Gender {MALE, FEMALE}

   private long id;
   private String name;
   private Gender gender;
   private LocalDate dob;
   private double income;

   public Programmer(long id, String name, Gender gender, LocalDate dob, double income) {
      this.id = id;
      this.name = name;
      this.gender = gender;
      this.dob = dob;
      this.income = income;
   }
         public long getId() {
         return id;
   }
         public void setId(long id) {
         this.id = id;
   }
```

```
        public String getName() {
       return name;
}

public void setName(String name) {
   this.name = name;
}

public Gender getGender() {
   return gender;
}

public void setGender(Gender gender) {
   this.gender = gender;
}

public LocalDate getDob() {
   return dob;
}

public void setDob(LocalDate dob) {
   this.dob = dob;
}

public double getIncome() {
   return income;
}

public void setIncome(double income) {
   this.income = income;
}

public boolean isMale() {
   return this.gender == Gender.MALE;
}

public boolean isFemale() {
   return this.gender == Gender.FEMALE;
}

@Override
public String toString() {
   return String.format("(%s, %s, %s, %s, %.2f)", id, name, gender, dob, income);
}
```

```
   public static List<Programmer> programmers() {
            Programmer Suman = new Programmer(1, "Suman", Gender.MALE, LocalDate.of(1943, Month.FEBRUARY, 4), 6000.0);
           Programmer Mahesh = new Programmer(2, "Mahesh", Gender.MALE, LocalDate.of(1969, Month.DECEMBER, 28), 7100.0);
       Programmer Kiranmayi = new Programmer(3, "Kiranmayi", Gender.FEMALE, LocalDate.of(1815, Month.DECEMBER, 10), 8700.0);
            Programmer David = new Programmer(4, "David", Gender.MALE, LocalDate.of(1938, Month.JANUARY, 10), 1800.0);
            Programmer Eliza = new Programmer(5, "Eliza", Gender.FEMALE, LocalDate.of(1955, Month.OCTOBER, 28), 2420.0);
            Programmer Rajesh = new Programmer(6, "Rajesh", Gender.MALE, LocalDate.of(1949, Month.FEBRUARY, 19), 12400.0);
      return Arrays.asList(Suman, Mahesh, Kiranmayi, David, Eliza, Rajesh);
   }
}
```

———

```
public void filteringOperation()
{
        IntStream.rangeClosed(1, 10)
                .filter( x -> x%2 == 0)
                .forEach(System.out::println);
 }
```

In the above sample rangeClosed method would pick numbers from 1 to 10 and filter only even numbers and prints each of them

```
public void filterProgrammers()
{
         //Function <Programmer,String> getProgrammerName = p -> p.getName();
        //Predicate <Programmer> isFemaleProgrammer = p ->p.isFemale();

        /*  Stream <String> progs1 = Programmer.programmers()
                .stream()
                .filter(Programmer::isFemale)
                .map(Programmer::getName);
```

```
                progs1.collect(Collectors.toList());*/

                Programmer.programmers()
                .stream()
                .filter(Programmer::isFemale)
                .map(Programmer::getName)
                .forEach(System.out::println);
  }
/*
```
Filter function filters Programmers i.e female Programmers and map function would get the Names of those programmers and collected into the List
```
*/
public void multipleProgrammers()
{
        Function <Programmer,String> getProgrammerName = p->p.getName();
        Predicate <Programmer> isMaleProgrammer = p->p.isMale();
        Predicate <Programmer> isEarningMoreThan5000 = p -> p.getIncome() > 5000;

Predicate <Programmer> isMaleAndEarningMoreThan5000 = p -> p.isMale() && p.getIncome() > 5000;
        Programmer.programmers()
        .stream()
        .filter(isEarningMoreThan5000)
        .filter(isMaleAndEarningMoreThan5000)
        .filter(Programmer::isMale)
        .map(Programmer :: getName)
        .forEach(System.out::println);}
```
/* Predicates check the conditionals, here above picks all Programmers if the gender is Male and Income > 5000; Predicates can store info based on the conditions
```
*/
public static void main(String[] args) {
  // TODO Auto-generated method stub
  FilteringOperations fop = new FilteringOperations();
        //  fop.filteringOperation();
        //  fop.filterProgrammers();
        fop.multipleProgrammers();
```

```
  }  //All the functions are to be enclosed in a Class
  // For example a class FilteringOperations
  // Which is not shown here in the Example
}
```

Function can define some functionalities like getName() etc and Predicates can define conditionals and filter can filter based on the Predicates & Functions, here above

getProgrammerName() is a **Function** and isMaleProgrammer & isEarningMoreThan5000 are **Predicates** which are further **filtered** and streamed through and **mapped** to get Names and finally printed on the Console.

Streams – Terminal Operations

- Terminal operations are responsible for giving the 'final' output for a Stream in operation, and in the process they terminate a Stream.
- Terminal Operations thus do not return a Stream as their output.
- Apart from returning a Stream, terminal operations can return any value, or even no value(void)

Stream Seration	**Goal**	**Input**
forEach	For every item, outputs something	Consumer
collect	Accumulates elements in a stream into a container	Collector
reduce	Accumulates elements and performs reduction of items	
count	Counts current items	

Streams – From Arrays -Terminal Operations :Sample1

```
    void integerArrayStream() {
```

```
        IntStream intStream = Arrays.stream(new int[]{1, 2, 3, 4, 5});
        intStream.forEach(System.out::println);
        }
        void stringArrayStream() {
                Stream <String> stringStream = Arrays.stream(new String[]{"I", "Me", "You"});
                stringStream.forEach(System.out::println);
        }
        public static void main(String... args) {
        //StreamsFromArrays is a class where this function is said to be in - which is not shown here
        StreamsFromArrays streamsFromArrays = new StreamsFromArrays();
        streamsFromArrays.integerArrayStream();
        System.out.println("-----------------------------------------");
        streamsFromArrays.stringArrayStream();
        }
```

In the above example an Array of stream holding strings is printed on the console.

```
        void createStringStream() {
        Set<String> pronouns = new HashSet<>();
          pronouns.add("I");
          pronouns.add("Me");
          pronouns.add("You");
      pronouns.stream().forEach(System.out::println);
        }
public static void main(String... args) {
        StreamsFromCollections streamsFromCollections = new   StreamsFromCollections();
       streamsFromCollections.createStringStream();
        }
```

//Here above a HashSet holding Strings is streamed that is displayed on the console.

Sample 2:

```
void iterate() {
        Stream<Long> naturalNumbers = Stream.iterate(1L, x -> x + 1).limit(10);
```

```
        // Show the usage of skip(long)
        Stream<Long> oddNaturalNumbers = Stream.iterate(1L, x -> x + 2).limit(10);
        naturalNumbers.forEach(x-> System.out.print(x + " "));
        oddNaturalNumbers.forEach(x-> System.out.print(x + " "));
        }
        void generate() {
                Stream.generate(Math::random).limit(10).forEach(x-> System.out.print(x+" "));
        }
        public static void main(String... args) {
                 iterate();  // You need to call through Objects of the Class, that part is not shown
                generate(); // You need to call through Objects of the Class,that part is not shown


        }
```

Output is as follows:

```
1
2
3
4
5
6
7
8
9
10
----------
1
3
5
7
9
11
13
15
17
19
----------
0.25119046133796186
0.8785150461206035
```

0.5551891352989232
0.5836705040562682
0.533073827194257
0.9916957067527586
0.8871981356241398
0.7002030158382563
0.13563778198719423
0.74532844537128

Reduction

A reduction operation (also called as fold) takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation.
Syntax
1.Optional<T> reduce(BinaryOperator<T> accumulator)
2.T reduce(T identity, BinaryOperator<T> accumulator)
3.<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)

Collector

Collector is a reducer operation.
Result may be one single collection or any type of one object instance
Gets the desired values of each item, process and returns a single result
Converts the stream into a collection (List, Set, Map).

Prebuilt Collector implementations

Collectors.toList()	Puts items into a list
Collectors.toCollection(TreeSet::new)	Puts items into a desired container. Container is supplied with a supplier
Collectors.joining(", ")	Joins multiple items into a single item by concatenating them
Collectors.summingInt(item::getAge)	Sums the values of each item with given supplier
Collectors.groupingBy()	Groups the items with given classifier and mapper
Collectors.partitioningBy()	Partitions the items with given predicate and mapper

Optional

A container object which may or may not contain a non-null value

Purpose of this class is to provide a type-level solution for representing optional values instead of null references.

Located at java.util.Optional<T>

Some of the Useful Methods:

public static <T> ptional<T> empty()	This method returns an empty Optional object. No value is present for this Optional.
public static <T> Optional<T> of(T value)	This method returns an Optional with the specified value that is not null.
public static <T> Optional<T> ofNullable(T value)	This method returns an Optional describing the specified value if the value is non-null; otherwise, it returns an empty Optional.
public T get()	If a value is present in this Optional, then it returns the value. Otherwise, it throws NoSuchElementException.
public boolean isPresent()	This method returns a true value if there is a value present. Otherwise, it returns false.
public void ifPresent(Consumer<? super T> consumer)	If a value is present, then the consumer with the provided value is invoked. Otherwise, it does nothing.
public T orElse(T other)	This method returns the value if present; otherwise, it returns other.
public T orElseGet(Supplier<? extends T> other)	This method returns the value if present. Otherwise, it invokes other and returns the result of the invocation.

DATE & TIME API of Java8

Java 8 Introduced new APIs for DATE and Time to address the shortcomings of the older java.util.Date and java.util.Calendar.

Java.util.Date API had few disadvantages like

- For example, the existing classes (such as java.util.Date and SimpleDateFormatter) aren't thread-safe, leading to potential concurrency issues for users—not something the average developer would expect to deal with when writing date-handling code.

- Some of the date and time classes also exhibit quite poor API design. For example, years in java.util.Date start at 1900, months start at 1, and days start at 0—not very intuitive.

In order to address these problems and provide better support in the JDK core, a new date and time API, which is free of these problems, has been designed for Java SE 8.

The new API is driven by three core ideas:

- **Immutable-value classes.** One of the serious weaknesses of the existing formatters in Java is that they aren't thread-safe. This puts the burden on developers to use them in a thread-safe manner and to think about concurrency problems in their day-to-day development of date-handling code. The new API avoids this issue by ensuring that all its core classes are immutable and represent well-defined values.
- **Domain-driven design.** The new API models its domain very precisely with classes that represent different use cases for Date and Time closely. This differs from previous Java libraries that were quite poor in that regard. For example, java.util.Date represents an instant on the timeline—a wrapper around the number of milli-seconds since the UNIX epoch—but if you call toString(), the result suggests that it has a time zone, causing confusion among developers.
- **Separation of chronologies.** The new API allows people to work with different calendaring systems in order to support the needs of the business.

Java8 Date and Time - LocalDate

The LocalDate represents a date in ISO format (yyyy-MM-dd) without time. We can use it to store dates like birthdays and paydays.
An instance of current date can be created from the system clock:

```
LocalDate localDate = LocalDate.now();
```

And we can get the LocalDate representing a specific day, month and year by using the of method or the parse method.
For example, these code snippets represent the LocalDate for February 20, 2015:

```
LocalDate.of(2015, 02, 20);
LocalDate.parse("2015-02-20");
```

The LocalDate provides various utility methods to obtain a variety of information. Let's have a quick peek at some of these API methods.

The following code snippet gets the current local date and adds one day:

```
LocalDate tomorrow = LocalDate.now().plusDays(1);
```

This example obtains the current date and subtracts one month. Note how it accepts an enum as the time unit:

```
LocalDate previousMonthSameDay = LocalDate.now().minus(1, ChronoUnit.MONTHS);
```

In the following two code examples, we parse the date "2016-06-12" and get the day of the week and the day of the month respectively. Note the return values — the first is an object representing the DayOfWeek, while the second is an int representing the ordinal value of the month:

```
DayOfWeek sunday = LocalDate.parse("2016-06-12").getDayOfWeek();

int twelve = LocalDate.parse("2016-06-12").getDayOfMonth();
```

We can test if a date occurs in a leap year, for example the current date:

```
boolean leapYear = LocalDate.now().isLeapYear();
```

Java8 Date and Time - LocalTime

- The LocalTime represents time without a date.
- Similar to LocalDate, we can create an instance of LocalTime from the system clock or by using parse and of methods.
- We'll now take a quick look at some of the commonly used APIs.
- An instance of current LocalTime can be created from the system clock:

```
LocalTime now = LocalTime.now();
```

- We can create a LocalTime representing 6:30 a.m. by parsing a string representation:

```
LocalTime sixThirty = LocalTime.parse("06:30");
```

- Let's create a LocalTime by parsing a string and adding an hour to it by using the "plus" API. The result would be LocalTime representing 7:30 a.m.:

```
LocalTime sevenThirty = LocalTime.parse("06:30").plus(1, ChronoUnit.HOURS);
```

Java8 Date and Time - LocalDate

Method	Description

LocalDateTime atTime(int hour, int minute)	It is used to combine this date with a time to create a LocalDateTime.
int compareTo(ChronoLocalDate other)	It is used to compares this date to another date.
boolean equals(Object obj)	It is used to check if this date is equal to another date.
String format(DateTimeFormatter formatter)	It is used to format this date using the specified formatter.
int get(TemporalField field)	It is used to get the value of the specified field from this date as an int.
boolean isLeapYear()	It is used to check if the year is a leap year, according to the ISO proleptic calendar system rules.
LocalDate minusDays(long daysToSubtract)	It is used to return a copy of this LocalDate with the specified number of days subtracted.
LocalDate minusMonths(long monthsToSubtract)	It is used to return a copy of this LocalDate with the specified number of months subtracted.
static LocalDate now()	It is used to obtain the current date from the system clock in the default time-zone.
LocalDate plusDays(long daysToAdd)	It is used to return a copy of this LocalDate with the specified number of days added.

LocalDate plusMonths(long monthsToAdd)	It is used to return a copy of this LocalDate with the specified number of months added.
LocalDate plusMonths(long monthsToAdd)	It is used to return a copy of this LocalDate with the specified number of months added.
int getDayOfMonth()	It gets the day-of-month field.
int getDayOfYear()	It gets the day-of-year field.
Month getMonth()	It gets the month-of-year field using the Month enum.
int getMonthValue()	It gets the month-of-year field from 1 to 12.
int getYear()	It gets the year field.
int lengthOfMonth()	It returns the length of the month represented by this date.
int lengthOfYear()	It returns the length of the year represented by this date.
static LocalDate ofYearDay(int year, int dayOfYear)	It obtains an instance of LocalDate from a year and day-of-year.
static LocalDate parse(CharSequence text)	It obtains an instance of LocalDate from a text string such as 2007-12-03
static LocalDate parse(CharSequence text, DateTimeFormatter formatter)	It obtains an instance of LocalDate from a text string using a specific formatter.

Java8 Date and Time - LocalTime

Method	Description
LocalDateTime atDate(LocalDate date)	It is used to combine this time with a date to create a LocalDateTime.
int compareTo(LocalTime other)	It is used to compare this time to another time.
String format(DateTimeFormatter formatter)	It is used to format this time using the specified formatter.
int get(TemporalField field)	It is used to get the value of the specified field from this time as an int.
LocalTime minusHours(long hoursToSubtract)	It is used to return a copy of this LocalTime with the specified number of hours subtracted.
LocalTime minusMinutes(long minutesToSubtract)	It is used to return a copy of this LocalTime with the specified number of minutes subtracted.
static LocalTime now()	It is used to obtain the current time from the system clock in the default time-zone.
static LocalTime of(int hour, int minute, int second)	It is used to obtain an instance of LocalTime from an hour, minute and second.
LocalTime plusHours(long hoursToAdd)	It is used to return a copy of this LocalTime with the specified number of hours added.
LocalTime plusMinutes(long minutesToAdd)	It is used to return a copy of this LocalTime with the specified number of minutes added.

----------EOD----------